

Object Oriented Design Patterns

<https://refactoring.guru/design-patterns/what-is-pattern>

- Creational
- Behaviourial
- Structural
- Architectural

Creational

Creational design patterns are used to

- Create objects effectively
- Add flexibility to software design

Many designs start by using [Factory Method](#) (less complicated and more customizable via subclasses) and evolve toward [Abstract Factory](#), [Prototype](#), or [Builder](#) (more flexible, but more complicated).

Factory Pattern

- **Object Creation without Exposing Concrete Classes:**
 - Factories provide a centralized way to create objects, allowing for better encapsulation and abstraction.
- **Dynamic Object Creation:**
 - Factories are useful when the exact class of the object to be created may vary at runtime based on certain conditions or configurations.
 - Factories can select the appropriate subclass or implementation to create based on these conditions.
- **Polymorphic Object Creation:**
 - Allowing clients to create objects without knowing the specific subclass or implementation being instantiated.
 - This promotes loose coupling and simplifies client code.
- **Difference to Dependency Injection:**
 - **When using a factory your code is still actually responsible for creating objects. By DI you outsource that responsibility to another class or a framework, which is separate from your code.**

```
class Burger:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def print(self):
        print(self.ingredients)

class BurgerFactory:
```

```
def createCheeseBurger(self):
    ingredients = ["bun", "cheese", "beef-patty"]
    return Burger(ingredients)

def createDeluxeCheeseBurger(self):
    ingredients = ["bun", "tomatoe", "lettuce", "cheese", "beef-patty"]
    return Burger(ingredients)

def createVeganBurger(self):
    ingredients = ["bun", "special-sauce", "veggie-patty"]
    return Burger(ingredients)

burgerFactory = BurgerFactory()
burgerFactory.createCheeseBurger().print()
burgerFactory.createDeluxeCheeseBurger().print()
burgerFactory.createVeganBurger().print()
```

Copy

Output

```
["bun", 'cheese', 'beef-patty']

['bun', 'tomatoe', 'lettuce', 'cheese', 'beef-patty']

['bun', 'special-sauce', 'veggie-patty']
```

Builder Pattern

1. Builder Pattern:

- **Complex Object Construction:**
 - Create complex objects with multiple optional or mandatory parameters, and the number of parameters makes constructor overloading impractical or confusing.
- **Fluent Interface:**
 - Allowing for a more expressive and readable way to construct objects, especially when chaining multiple method calls together.
- **Flexible Object Creation:**

- Create objects in a step-by-step manner, allowing for more flexibility in the construction process.
- When different parts of the object need to be configured independently or when the construction process is dynamic.

- **Parameter Omission:**

- Allow parameters to be omitted or set to default values, providing more control over object creation and reducing the need for multiple overloaded constructors

```
class Burger:
```

```
    def __init__(self):
```

```
        self.buns = None
```

```
        self.patty = None
```

```
        self.cheese = None
```

```
    def setBuns(self, bunStyle):
```

```
        self.buns = bunStyle
```

```
    def setPatty(self, pattyStyle):
```

```
        self.patty = pattyStyle
```

```
    def setCheese(self, cheeseStyle):
```

```
        self.cheese = cheeseStyle
```

```
class BurgerBuilder:
```

```
    def __init__(self):
```

```
        self.burger = Burger()
```

```
    def addBuns(self, bunStyle):
```

```
        self.burger.setBuns(bunStyle)
```

```
        return self
```

```
    def addPatty(self, pattyStyle):
```

```
        self.burger.setPatty(pattyStyle)
```

```
        return self
```

```
    def addCheese(self, cheeseStyle):
```

```
        self.burger.setCheese(cheeseStyle)
```

```
        return self
```

```
    def build(self):
```

```
return self.burger
```

```
burger = BurgerBuilder() \
    .addBuns("sesame") \
    .addPatty("fish-patty") \
    .addCheese("swiss cheese") \
    .build()
```

Singleton

Singleton Pattern is considered unpythonic

<https://www.youtube.com/watch?v=Rm4JP7JfsKY&t=634s>

Why it is bad:

- If you inherit from it, you can get multiple instances, which shouldn't be allowed.
- Testing code is hard with singleton because you cannot create multiple fresh instances for testing
- Does not work well with multi threaded applications because raise condition of

```
class ApplicationState:
    instance = None

    def __init__(self):
        self.isLoggedIn = False

    @staticmethod
    def getAppState():
        if not ApplicationState.instance:
            ApplicationState.instance = ApplicationState()
        return ApplicationState.instance

appState1 = ApplicationState.getAppState()
print(appState1.isLoggedIn)

appState2 = ApplicationState.getAppState()
appState1.isLoggedIn = True
```

```
print(appState1.isLoggedIn)
print(appState2.isLoggedIn)
```

```
>> False
```

```
>> True
```

```
>> True
```

Better Methodologies than Singleton:

- Object Pool Pattern manages a fixed number of instances instead of one,
 - ex. Managing database connections of graphics objects with a lot of data drawn over and over again
- Global Object pattern, as in the Constant pattern, a module instantiates an object at import time and assigns it a name in the module's global scope.
 - But the object does not simply serve as data; it is not merely an integer, string, or data structure. Instead, the object is made available for the sake of the methods it offers — for the actions it can perform.
- Python's module system and the fact that modules are imported only once make it easy to implement singleton behavior using modules.

Behaviourial

Observer / PubSub

It's common for different components of an app to respond to events or state changes, but how can we communicate these events?

The Observer pattern is a popular solution. We have a Subject (aka Publisher) which will be the source of events. And we could have multiple Observers (aka Subscribers) which will receive events from the Subject in realtime.

```
class YoutubeChannel:
    def __init__(self, name):
        self.name = name
        self.subscribers = []

    def subscribe(self, sub):
        self.subscribers.append(sub)

    def notify(self, event):
        for sub in self.subscribers:
            sub.sendNotification(self.name, event)

from abc import ABC, abstractmethod

class YoutubeSubscriber(ABC):
    @abstractmethod
    def sendNotification(self, event):
        pass

class YoutubeUser(YoutubeSubscriber):
    def __init__(self, name):
        self.name = name

    def sendNotification(self, channel, event):
        print(f"User {self.name} received notification from {channel}: {event}")

channel = YoutubeChannel("neetcode")
```

```
channel.subscribe(YoutubeUser("sub1"))
channel.subscribe(YoutubeUser("sub2"))
channel.subscribe(YoutubeUser("sub3"))

channel.notify("A new video released")
```

Copy

In this case we have multiple Subscribers listening to a single published. But users could also be subscribed to multiple channels. Since the Publishers & Subscribers don't have to worry about each others' implementations, they are loosely coupled.



```
“ User sub1 received notification from neetcode: A new video released

User sub2 received notification from neetcode: A new video released

User sub3 received notification from neetcode: A new video released
```


Iterator

Many objects in python have built-in iterators. That's why we can conveniently iterate through an array using the key word `in`.

```
myList = [1, 2, 3]
for n in myList:
    print(n)
```

Copy

Output



```
“ 1

  2

  3
```


For more complex objects, like Linked Lists or Binary Search Trees, we can define our own iterators.

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

class LinkedList:
    def __init__(self, head):
        self.head = head
        self.cur = None

    # Define Iterator
    def __iter__(self):
        self.cur = self.head
        return self

    # Iterate
    def __next__(self):
        if self.cur:
            val = self.cur.val
            self.cur = self.cur.next
            return val
        else:
            raise StopIteration

# Initialize LinkedList
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
myList = LinkedList(head)

# Iterate through LinkedList
for n in myList:
    print(n)
```

Copy

Output

|

1

2

3

Strategy

A Class may have different behaviour, or invoke a different method based on something we define (i.e. a Strategy). For example, we can filter an array by removing positive values; or we could filter it by removing all odd values. These are two filtering strategies we could implement, but we could add many more.

```
from abc import ABC, abstractmethod

class FilterStrategy(ABC):

    @abstractmethod
    def removeValue(self, val):
        pass

class RemoveNegativeStrategy(FilterStrategy):

    def removeValue(self, val):
        return val < 0

class RemoveOddStrategy(FilterStrategy):

    def removeValue(self, val):
        return abs(val) % 2

class Values:

    def __init__(self, vals):
        self.vals = vals

    def filter(self, strategy):
        res = []
        for n in self.vals:
```

```
        if not strategy.removeValue(n):
            res.append(n)
    return res
```

```
values = Values([-7, -4, -1, 0, 2, 6, 9])
```

```
print(values.filter(RemoveNegativeStrategy()))
print(values.filter(RemoveOddStrategy()))
```

Copy

Output

```
“ [0, 2, 6, 9]
```

```
[-4, 0, 2, 6]
```

A common alternative to this pattern is to simply pass in an inline / lambda function, which allows us to extend the behaviour of a method or class.

State Machines

When to use it?

State management is often part of a framework already, so building from scratch is rare. Different classes such as below can denote different states, but more pythonic way is to use different modules and functions, where each module can denote a different state. Can be used in such example like a "simple" and "advanced" mode of a GUI.

```
from dataclasses import dataclass
from typing import Protocol
```

```
class DocumentState(Protocol):
    def edit(self):
        ...

    def review(self):
```

```
...
```

```
def finalize(self):
```

```
...
```

```
class DocumentContext(Protocol):
```

```
    content: list[str]
```

```
    def set_state(self, state: DocumentState) -> None:
```

```
        ...
```

```
    def edit(self):
```

```
        ...
```

```
    def review(self):
```

```
        ...
```

```
    def finalize(self):
```

```
        ...
```

```
    def show_content(self):
```

```
        ...
```

```
@dataclass
```

```
class Draft:
```

```
    document: DocumentContext
```

```
    def edit(self):
```

```
        print("Editing the document...")
```

```
        self.document.content.append("Edited content.")
```

```
    def review(self):
```

```
        print("The document is now under review.")
```

```
        self.document.set_state(Reviewed(self.document))
```

```
    def finalize(self):
```

```
        print("You need to review the document before finalizing.")
```

```
@dataclass
class Reviewed:
    document: DocumentContext

    def edit(self):
        print("The document is under review, cannot edit now.")

    def review(self):
        print("The document is already reviewed.")

    def finalize(self):
        print("Finalizing the document...")
        self.document.set_state(Finalized(self.document))
```

```
@dataclass
class Finalized:
    document: DocumentContext

    def edit(self):
        print("The document is finalized. Editing is not allowed.")

    def review(self):
        print("The document is finalized. Review is not possible.")

    def finalize(self):
        print("The document is already finalized.")
```

```
class Document:
    def __init__(self):
        self.state: DocumentState = Draft(self)
        self.content: list[str] = []

    def set_state(self, state: DocumentState):
        self.state = state

    def edit(self):
        self.state.edit()
```

```
def review(self):
    self.state.review()

def finalize(self):
    self.state.finalize()

def show_content(self):
    print("Document content:", " ".join(self.content))
```

```
def main() -> None:
    document = Document()

    document.edit() # Expected: "Editing the document..."
    document.show_content() # Expected: "Document content: Edited content."
    document.finalize() # Expected: "You need to review the document before finalizing."
    document.review() # Expected: "The document is now under review."
    document.edit() # Expected: "The document is under review, cannot edit now."
    document.finalize() # Expected: "Finalizing the document..."
    document.edit() # Expected: "The document is finalized. Editing is not allowed."

if __name__ == "__main__":
    main()
```

Structural

Structural Patterns

Focused on how objects and classes can be composed to form larger structures while keeping these structures flexible and efficient. These patterns deal with object composition and relationships between classes or objects. They help in building a flexible and reusable codebase by promoting better organization and modularity.

Facade

According to Oxford Languages, a Facade is

“an outward appearance that is maintained to conceal a less pleasant or creditable reality.”

In the programming world, the "outward appearance" is the class or interface we interact with as programmers. And the "less pleasant reality" is the complexity that is hidden from us.

So a Facade, is simply a wrapper class that can be used to abstract lower-level details that we don't want to worry about.

```
# Python arrays are dynamic by default, but this is an example of resizing.
```

```
class Array:
```

```
    def __init__(self):
```

```
        self.capacity = 2
```

```
        self.length = 0
```

```
        self.arr = [0] * 2 # Array of capacity = 2
```

```
# Insert n in the last position of the array
```

```
def pushback(self, n):
```

```
    if self.length == self.capacity:
```

```
        self.resize()
```

```
# insert at next empty position
```

```
self.arr[self.length] = n
```

```

self.length += 1

def resize(self):
    # Create new array of double capacity
    self.capacity = 2 * self.capacity
    newArr = [0] * self.capacity

    # Copy elements to newArr
    for i in range(self.length):
        newArr[i] = self.arr[i]
    self.arr = newArr

# Remove the last element in the array
def popback(self):
    if self.length > 0:
        self.length -= 1

```

Copy

Adapter Pattern (Wrapper)

Adapter Pattern allow incompatible objects to be used together.

This supports *Composition over Inheritance* and *Open-Closed Principle*. Instead of modifying the base class, we extend the class behavior

Adapter is used when refactoring code, would never consider an adapter pattern from the start.

If a `MicroUsbCable` class is initially incompatible with `UsbPort`, we can create a adapter/wrapper class, which makes them compatible. In this case, a `MicroToUsbAdapter` makes them compatible, similar to how we use adapters in the real-world.

```

class UsbCable:
    def __init__(self):
        self.isPlugged = False

    def plugUsb(self):
        self.isPlugged = True

class UsbPort:
    def __init__(self):

```



```
self.portAvailable = True

def plug(self, usb):
    if self.portAvailable:
        usb.plugUsb()
        self.portAvailable = False

# UsbCables can plug directly into Usb ports
usbCable = UsbCable()
usbPort1 = UsbPort()
usbPort1.plug(usbCable)

class MicroUsbCable:
    def __init__(self):
        self.isPlugged = False

    def plugMicroUsb(self):
        self.isPlugged = True

class MicroToUsbAdapter(UsbCable):
    def __init__(self, microUsbCable):
        self.microUsbCable = microUsbCable
        self.microUsbCable.plugMicroUsb()

# can override UsbCable.plugUsb() if needed

# MicroUsbCables can plug into Usb ports via an adapter
microToUsbAdapter = MicroToUsbAdapter(MicroUsbCable())
usbPort2 = UsbPort()
usbPort2.plug(microToUsbAdapter)
```

Architectural

Monolith

A monolithic architecture is an architectural pattern where all components of an application are combined into a single codebase and deployed as a single unit. In a monolithic design pattern:

1. **Codebase:** All the application's functionalities, including the user interface, business logic, and data access code, are packaged together into a single codebase.
2. **Deployment:** The entire application is deployed as a single unit, typically on a single server or a set of servers.
3. **Communication:** Communication between different components of the application typically occurs through function calls or method invocations within the same process.
4. **Scalability:** Scaling a monolithic application involves scaling the entire application, which can be challenging as it requires replicating the entire application stack.

Monolithic architectures are often simpler to develop and deploy initially, especially for small to medium-sized applications. They offer advantages such as ease of development, debugging, and testing, as well as simpler deployment and management.

However, as applications grow in complexity and scale, monolithic architectures can become harder to maintain and scale. They may suffer from issues such as tight coupling between components, limited scalability, and difficulty in adopting new technologies or frameworks.

In contrast, microservices architecture, where an application is decomposed into smaller, independently deployable services, offers benefits such as improved scalability, flexibility, and maintainability. However, microservices come with their own set of challenges, such as increased complexity in managing distributed systems and communication overhead between services.

Ultimately, the choice between monolithic and microservices architectures depends on factors such as the size and complexity of the application, scalability requirements, development team expertise, and organizational constraints.

Model View Controller

The MVC architecture is very broad and can change depending on the programming language and type of application you are doing, so in this case, yes your approach can be accepted as correct.

What I have learned from static typed languages is that you define the model and views as complete separate entities, and the controller takes an instance of both model and views as parameters.

What you need to ask yourself to define if your app is MVC is the following:

- If I change something in the view do I break anything in the model?
- If I change something in the model do I break anything in the view?
- Is the controller communicating everything in both view and model so that they don't have to communicate with each other?

If nothing breaks and the controller does all of the communication then yes, your application is MVC.

You might want to look into design patterns such as Singleton, Factory and others that all use the MVC architecture and define ways to implement it.

Microservices

Others

Event-Driven Architecture (EDA) and Service-Oriented Architecture (SOA) are both architectural patterns used in software design, but they have different approaches and focus areas. Here's a comparison between the two:

1. **Communication Paradigm:**

- **EDA:** In Event-Driven Architecture, communication between components is based on events. Components publish events when certain actions or changes occur, and other components subscribe to these events and react accordingly. This asynchronous communication model allows for loose coupling between components and enables real-time responsiveness.
- **SOA:** In Service-Oriented Architecture, communication between components is typically based on service interfaces. Services expose their functionality through well-defined interfaces (APIs) that other services or clients can invoke. Communication in SOA is often synchronous, although asynchronous messaging can also be used.

2. **Granularity:**

- **EDA:** EDA tends to be more fine-grained, with events representing specific actions or changes within the system. Components can react to individual events and perform specific actions accordingly.
- **SOA:** SOA can vary in granularity, but services tend to encapsulate larger units of functionality. Services are typically designed to represent business capabilities or domain entities, providing coarse-grained operations.

3. **Flexibility:**

- **EDA:** EDA offers greater flexibility and agility, as components can react to events in a dynamic and decentralized manner. New components can be added or existing components modified without affecting the overall system.
- **SOA:** SOA also provides flexibility, but to a lesser extent compared to EDA. Changes to services may require coordination between service providers and consumers, and service contracts need to be carefully managed to ensure compatibility.

4. **Scalability:**

- **EDA:** EDA inherently supports scalability, as components can be scaled independently based on event processing requirements. Event-driven systems can handle bursts of traffic more gracefully by distributing processing across multiple instances.
- **SOA:** SOA can be scalable, but scaling individual services may not always be straightforward, especially if services have dependencies or shared resources.

5. **Complexity:**

- **EDA:** EDA can introduce complexity, particularly in managing event propagation, event schemas, and ensuring event consistency across components. Event-driven systems may also require additional infrastructure for event processing and management.
- **SOA:** SOA tends to be less complex compared to EDA, as services have well-defined interfaces and interactions. However, managing service contracts, versioning, and service discovery can still introduce complexity, especially in large-scale deployments.

In summary, Event-Driven Architecture is well-suited for scenarios where real-time responsiveness, loose coupling, and flexibility are paramount, while Service-Oriented Architecture provides a more structured approach to building distributed systems with reusable and interoperable services. The choice between EDA and SOA depends on the specific requirements, constraints, and trade-offs of the application or system being designed.